

Programming and Performance on a Cube-Connected Architecture

*John Gustafson
Gary Montry*

**Sandia National Laboratories
Albuquerque, NM 87185**

ABSTRACT

The first generation of commercial hypercube⁶ multiprocessors is now over two years old. We have discovered the weaknesses of these machines and learned to program around them; also, we have discovered their strengths and utilized them to our advantage. This discussion will focus on the strengths and weaknesses of this first generation of machines and their effects on meaningful software development and implementation. In particular, we will address these issues with respect to hypercubes of *a thousand or more* processing elements.

INTRODUCTION

The issues discussed here pertain to the authors' experiences with a limited class of application programs: fluid dynamics, factoring large numbers, structural analysis, sieving prime numbers, and electrostatics. Hence, these discussions only represent a subset of the vast range of applications that might be put on a large computer. However, improvements in architecture suggested by some classes of applications will generally improve performance of the computer over a much wider range of programs. The observations in this paper are for problems that are

- Fairly regular in space decomposition
- Explicit in stepping through time or iterations
- Decomposable with static methods
- Parallel enough for a 1024-processor ensemble

The large number of processors forces one to consider *every sequential aspect* of a run, including loading the program and viewing results. Unlike systems with, say, ten or fewer processors, the 1024-processor environment is very unforgiving of old-fashioned serial programming habits. Some of the issues presented here stem from the hypercube interconnect; others are the result of massive ensemble parallelism generally.

HYPERCUBES AT SANDIA

Sandia obtained a completely functioning 10-dimensional (1024 processors) hypercube

on November 20, 1987. There are also a number of *development systems*, low-order hypercubes based on personal computers, which are compatible with the large system. The Sandia hypercube environment is shown in Figure 1.

Figure 1. Sandia Hypercube Environment

The NCUBE hypercubes⁴ currently use an Intel 80286 host, the same microprocessor used in the IBM[®] PC-AT; however, the processor does not run in 8088-compatibility mode or under MS-DOS[®]. Both the large system and the development systems run under a UNIX-style operating system that controls the multitasking and system device resources. The hypercube is accessed from the host operating system just like any other device, via the `/dev` directory.

The NCUBE processors are proprietary, highly-integrated chips that are compatible with the 80286 host only in that they have the same storage format for floating-point and integer data. The NCUBE node resembles a VAX-11/780[®] with Floating Point Accelerator, in both architecture and performance. Each node has 512 KBytes of memory, of which 480 KBytes is available for program and data (depending on the size of communication buffers).

THE ENSEMBLE PARADIGM

Large ensembles should be used on large problems. This simple principle is widely ignored in the research community, which tends to seek parallel solutions and evaluate parallel performance for fixed-size problems. The issue is illustrated in Figure 2.

Figure 2. Ensemble Computing Performance Pattern

To keep the arithmetic hardware as busy as possible, one seeks to minimize the percent of time spent in interprocessor communication. This implies that the problem should use as much local memory as possible, to keep most of the communications internal. If the number of processors is doubled, *the size of the problem should be doubled as well*, so as to preserve this optimum use of memory. For years, however, “efficiency” has been defined as the speedup divided by the number of processors, for a fixed-size problem. This is the “Research Line” shown in Figure 2.

RESULTS TO DATE

We recognize the “Research Line” as the region of some academic interest to the computer science community; however, as a practical matter, the real strength in

massively parallel systems lies in their ability to do very large problems beyond the reach of conventional machines, not fixed problems in shorter amounts of time. This performance view is illustrated by the “Optimum Performance” line in Figure 2.

Along the “Research Line,” we have recently achieved the following speedups over the serial algorithms on four applications of importance to Sandia, using all 1024 processors:

Beam Bending (Finite Elements) using Conjugate Gradients	>350x
Nonlinear Second-Order CFD using the Flux-Corrected Transport method	>450x
Acoustic Wave Propagation using Explicit Finite Differences	>600x
Finding All Primes in $[1, 2 \times 10^6]$ using the Sieve of Eratosthenes	>600x

Our initial results along the “Optimum Performance” line have been even more encouraging. The conjugate gradient solver, when the number of degrees of freedom is scaled to the number of processors, essentially runs a thousand times faster on a thousand nodes compared to a hypothetical single processor with memory large enough to run such a problem. We refer to this as “scaled speedup.”

Figure 3. Scaled Performance — Structural Analysis

Similar performance curves have been obtained for the Wave Equation and Computational Fluid Dynamics.

PROGRAM LOADING

One must attend to *program loading* on distributed memory machines with a little more diligence than on their shared memory counterparts. A hypercube with only a single data path between the mass storage device and the ensemble of processors must ensure that the program load uses much of the available bandwidth between the host and the hypercube. For example, a modest-sized executable file of 100 KBytes loaded into the full 1024 processors requires the (redundant) loading of more than 100 MBytes of data. In order to accomplish this feat gracefully, one *must* take advantage of the fact that a binary spanning tree (requiring time of order $\log_2 N$, where N is the number of processors) is a subset of the hypercube interconnects. The difference between an order N and an order $\log_2 N$ algorithm is a factor of 100 on our hypercube! Loading from the host, one processor at a time, takes several minutes on our system. This worst-case situation

need only occur when a different program is loaded onto every node, a rare situation indeed. Most of our applications use exactly the same program on every node, permitting a logarithmic fan-out as shown in Figure 4 for a set of 16 processors labeled 0000 to 1111.

Figure 4. Logarithmic Fan-Out

This approach makes it possible to load programs into the entire hypercube in a few seconds.

The program load time for parallel processors might not seem like an important factor. But it is not unusual to want to run a job that involves only 10 billion floating-point operations, which would take about a minute and a half on the hypercube for the actual computation. It is clearly a waste of resource if the program load takes twice as long as its execution.

In addition to loading, the logarithmic communication concept is necessary for a variety of functions during program execution. For example, if an inner product must be computed across all processors, or a maximum element found, or a vote must be taken as to whether a technique has converged to a solution, or answers consolidated for output, then there is an order N technique that burdens the host or an order $\log_2 N$ technique that uses the nodes with a hundred times more parallelism. This is an important concept for multi-processors generally; even if the application appears to need only toroidal interconnect, massive parallelism will be degraded unless the ensemble has a fast binary (or higher) tree in its interconnect.

LANGUAGE

The NCUBE has Fortran, C, and assembler, for both the 80286 host and the processing nodes. In this first-generation machine, the compilers are obviously less mature regarding optimization than on a machine such as a VAX, so there is still reason to code critical sections of a program in assembler for improved performance. Many excellent features of the NCUBE processor (automatic constant-stride addressing, repeat-and-decrement instructions, floating-point hardware for argument reduction) remain inaccessible from C and Fortran at the time of writing.

Programming the ensemble involves writing *two* programs: one for the host and (at least) one for the hypercube. [If sophisticated graphics output is desired at high speed, one must also write a program for nodes dedicated to graphics I/O... or load existing graphics routines.] The program for the host, ideally, does little more than fetch the program and initial data from disk, load it onto the nodes, receive results, and display or store those results. Even serial parts of the application are best handled by the nodes rather than the host, since an individual node is about five times faster than the host microprocessor. The host is simply the hub of all the devices in the system. The node program, which need not be in the same language as the host program, resembles the non-I/O part of a

conventional application program.

Communication between nodes is handled with simple subroutine calls. It is this point that causes the greatest confusion about what it means to program a hypercube. There is no need to extend the language syntax itself. Hypercube communication uses a half-dozen calls for sending and receiving messages that are quickly learned and in no way revolutionize the language.

DEBUGGING

While developing programs on hypercubes, the most common mode of failure is that the system deadlocks as the result of failed interprocessor communication. There is no automatic reporting of the cause of the lockup. It is therefore *essential* that the system possess a debugger capable of interrogating the ensemble either globally or on a node-by-node basis. This is probably the most important piece of support software that a vendor can supply.

COMMUNICATIONS

Several shortcomings in the communication protocols of the first generation of cubes need to be addressed in the next generation.

The first of these weaknesses is unnecessary overhead, not just for message startup, but *during the actual transfer*. For a typical domain-decomposition problem, a message is *quadruple-buffered*, which cuts communication speed by several times. If a two-dimensional subdomain is stored in typical lexicographic order, then either the left-right or the top-bottom edges will not be stored contiguously. The current generation of hypercubes requires that a message be in a contiguous block of memory, so the edge must first be copied to a contiguous buffer before it is sent. The operating system call to write the message does *not* move that buffer, but first copies it to a dedicated area in system memory, and then returns control to the program. The reason for this is that one might otherwise alter the data while it is being sent, resulting in hard-to-detect and hard-to-repeat bugs. On the receiving end, the process is duplicated in reverse (See Figure 5).

Figure 5. Quadruple Buffering

The gathering and scattering of the vector can be eliminated with hardware/software support for constant-stride DMA. The move to and from system buffers should be an option that can be disabled for higher performance once bugs have been otherwise eliminated.

A related communications efficiency issue is the availability of truly overlapped internode communications. The hooks for non-blocking reads are available in the software for these machines, but have yet to be fully implemented. The buffering shown

in Figure 5 also means that at least half the time of the communication is spent using the processor to do memory-to-memory moves, and hence it is very difficult to get more than two DMA writes operating simultaneously. It is possible in principle for hypercubes to *completely overlap* their communication with the computation for most applications, which will further improve efficiency for all sizes of hypercubes. Note that if a separate processing entity (such as a simple finite-state machine) is provided to handle the communication, then no communication cost is incurred by the node processor itself except for the subroutine call. This paradigm appears to us to be highly desirable for cubes with thousands of processing elements.

The current generation of hypercubes is notorious for the relatively large startup times of message transmission. Much of this is caused by handshaking and routing protocol. We look forward to leaner protocol that recognizes nearest-neighbor connections only, and to hardware assistance in reducing message startup time. The high latency forces a programming style, at least in Fortran, that is somewhat opaque... using EQUIVALENCE statements to consolidate data wherever possible, changing the order of algorithm steps to batch messages together, and similar tricks.

ALGORITHM ISSUES

Large ensemble parallel processors greatly extend the applicable range of iterative methods and explicit methods for the solution of PDEs. Generally, explicit application programs can be written that require only one global operation per time step. The remaining interprocessor communications can be reduced to nearest-neighbor data exchanges within the hypercube. The number of data exchanges between adjacent nodes can be greatly reduced by data structure rearrangement (see the preceding section). Explicit methods are local, load-balanced, low in synchronization and communication cost, and clearly the method of choice for the first generation of hypercubes.

The first generation of hypercube multiprocessors does not seem well-suited to *direct solvers* that arise, for example, in discretized representations of partial differential equations¹. Direct inversion of matrices appears to require a large ratio of off-node communication to computation, regardless of the algorithm used. The direct solvers in use today are rife with sequential bottlenecks and non-local communication patterns. For example, the pivoting step makes all processors wait while a maximum is found and then the pivot row or column distributed (both using logarithmic complexity methods at best). Therefore, we expect machines that support medium to large parallel granularity to perform these tasks rather inefficiently.

LOAD BALANCE

Load balance on *shared memory* multiprocessors can be maintained by static decomposition of the computational domain or via the dynamic method of “self-scheduling,” where units of work are parceled out centrally on a first-come, first-served basis.

There is no reason that dynamic self-scheduling cannot be used on distributed memory machines, except that the global communication implied by a central scheduling entity is not efficient on a machine with high interprocessor message latency (on the order of a millisecond). As a result, research into the decomposition of irregular domains has focused on new techniques for determining optimized *static* decomposition of the problem. These techniques include simulated annealing³ and neural networks⁷; they are typically too expensive to use at run time.

The techniques necessary to efficiently load balance problems with data-dependent computational and communication loads, such as adaptive grids and Lagrangian (moving grid) dynamics, are still open for intensive research.

SUMMARY

The first generation of hypercubes has proved to be useful on a selected set of applications when more than 1000 processors are used. However, our work on large hypercubes illustrates deficiencies in the hardware and software that suggest improvements for the next generation. None of these deficiencies are inherent in the hypercube approach or technically difficult to correct; they simply require attention.

REFERENCES

- (1) Benner, R. E., Montry, G. R., and Weigand, G. C., "Concurrent Multifrontal Methods: Shared Memory, Cache, and Frontwidth Issues," *Int. Journal of Supercomputer Applications*, Vol. 1, No. 3, Fall 1987, pp. 26-44.
- (2) Fox, G. C., et al., *Solving Problems on Concurrent Processors*, 1987, (in publication), California Institute of Technology, 1987.
- (3) Laarhoven, J. M., and Aarts, E. H. C., *Simulated Annealing: Theory and Application*, D. Reidel Publishing Co. 1987.
- (4) *NCUBE Users Handbook*, NCUBE Corporation, Beaverton, Oregon, 1987.
- (5) Saad, Y., and Schultz, M. H., "Topological Properties of Hypercubes," *Research Report YALEU/DCS/RR-389*, June 1985.
- (6) Seitz, C., "The Cosmic Cube," *Communications of the ACM*, January 1985, Vol. 28, No. 1, pp. 22-33.
- (7) Williams, R. E., "Optimization by Computational Neural Nets," *CALT-68-1409*, California Institute of Technology, September, 1986.